The TVL Specification

Classen, Andreas; Boucher, Quentin; Faber, Paul; Heymans, Patrick

Publication date:
2010

Document Version
Early version, also known as pre-print

PReCISE – FUNDP
University of Namur
Rue Grandgagnage, 21
B-5000 Namur
Belgium

TECHNICAL REPORT October 24, 2010

AUTHORS A. Classen, Q. Boucher, P. Faber, P. Heymans
APPROVED BY P. Heymans
EMAILS {acs,qbo,pfaber,phe}@info.fundp.ac.be
STATUS Extended version of a paper appearing in the proceedings of the *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10)*, Linz, Austria.
REFERENCE P-CS-TR SPLBT-00000003
PROJECT MoVES
FUNDING FNRS, the Walloon Region, Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy

The TVL Specification

The TVL Specification*

Andreas Classen†, Quentin Boucher, Paul Faber and Patrick Heymans

PReCISE Research Centre
Faculty of Computer Science
University of Namur
5000 Namur, Belgium
Email: {acs,qbo,pfaber,phe}@info.fundp.ac.be

October 24, 2010

Abstract

In the scientific community, feature models are the de-facto standard for representing variability in software product line engineering. This is different from industrial settings where they appear to be used much less frequently. We have and other authors have argued that feature models lack concision, naturalness and expressiveness. Feature attributes, although an efficient tool for making models intuitive and concise, are not well understood and most existing notations do not support them at all. The graphical nature of feature models' syntax also appears to be a barrier to industrial adoption. Finally, existing tool support for graphical feature models is lacking or inadequate, and inferior in many regards to tool support for text-based formats.

TVL, a text-based feature modelling language, was designed specifically to address these shortcomings. In terms of expressiveness, TVL subsumes most existing dialects. The main goal of designing TVL was to provide engineers with a human-readable language with a rich syntax to make modelling easy and models natural, but also with a formal semantics to avoid ambiguity and allow powerful automations.

This report serves as the complete TVL specification and reference, providing its syntax and semantics in every detail.

1 Introduction

Software product line engineering (SPLE) is an increasingly popular software engineering paradigm which advocates systematic reuse across the software life-

*Previously titled *"Syntax and Semantics of TVL, a Text-based Feature Modelling Language"*. This text is not meant to be an introduction to TVL, rather a comprehensive reference. For a detailed introduction to TVL, the interested reader is referred to [1]; for a brief overview, to [2].

†FNRS Research Fellow

cycle. Central to the SPLE paradigm is the modelling and management of *variability*, i.e. *"the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts"* [3]. Variability is typically expressed in terms of *features*, i.e. first-class abstractions that shape the reasoning of the engineers and other stakeholders [4].

A set of features can be seen as the specification of a particular product of the product line (PL). Feature models (FMs) [5,6] delimit the set of valid products of the PL. FMs are directed acyclic graphs, generally trees, whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. The meaning of a decomposition link is that, if the parent feature is part of a product, then *some* of its child features have to be part of the product as well. Exactly which and how many of the child features have to be part depends on the type of the decomposition link. Consider the example FD in Figure 1 modelling a product line of personal computers. The *Computer* consists of a *Motherboard*, a *CPU*, a *Graphic Card* and some *Accessories*, which are optional (indicated by the hollow circle); all of these features are then further decomposed. In addition, although not shown in the figure, each of the features has a *price*, which can be modelled as an attribute, a typed parameter attached to each feature [7].

In the scientific community, FMs are the *de-facto* standard for representing the variability of an SPL. Several sources—our industry partners, discussions at the 2010 variability modelling (VaMoS) workshop [8] as well as recent literature reviews [9,10]—suggest that in the industrial world, in contrast, FMs appear to be used rarely.

Reasons for this, we believe, are their lack of conciseness and naturalness when it comes to modelling realistic SPLs and the graphical nature of their syntax [1]. Feature attributes, for instance, can be an efficient means to reduce the size and complexity of a FM. Yet, the semantics of FMs with attributes is not well understood and most existing notations and tools do not support them at all. The graphical syntax further constitutes a psychological barrier for engineers (having to draw models is deemed tedious and cumbersome) and poses a tooling problem. Existing tools for graphical FMs are generally research prototypes and are inferior in many regards to tool support for text-based formats (viz. text editors, source control systems, diff tools, no opaque file formats and so on).

To overcome these shortcomings, we designed `TVL` (Textual Variability Language), a text-based FM language. In terms of expressiveness, `TVL` subsumes most existing dialects. The main goal of designing `TVL` was to provide engineers with a human-readable language with a rich syntax to make modelling easy and models natural, but also with a formal semantics to avoid ambiguity and allow powerful automations. Further goals for `TVL` were to be *lightweight* (in contrast to the verbosity of XML for instance) and to be *scalable* by offering mechanisms for structuring the FM in various ways.

`TVL` is defined formally: its concrete, C-like, syntax is described by an LALR grammar, but it also has a mathematical abstract syntax and a denotational semantics. Having a well-defined tool-independent semantics further distinguishes `TVL` from most existing languages. A formal semantics allows anyone to imple-

ment the language, and this report contains all the information that is needed to accomplish this. A reference implementation including a parser and a reasoning library is available online.[1]

This report serves as the full specification for TVL. The journal paper [1] provides a more gentle introduction to TVL as well as a survey of existing textual variability modelling languages and other related work. It also presents two evaluations, one industrial and one comparative. The detail of the industrial evaluation was published separately by Hubaux *et al.* [11]. Since the journal paper also covers the semantics of TVL, it shares some content with this report.

The report is structured as follows: Section 2 introduces the TVL syntax with example code snippets before the formal EBNF grammar is given in Section 3. Section 4 covers well-formedness rules for TVL models. The formal semantics follows in Section 5.

2 Syntax

In this section we present an overview of the TVL syntax using code snippets before giving the formal BNF grammar. The following sub-sections introduce five major parts of the language: features, attributes, expressions, constraints and modularisation mechanisms.

The different concepts of TVL will be illustrated using a basic personal computer product family example FD presented in Section 1 and visible in Figure 1.

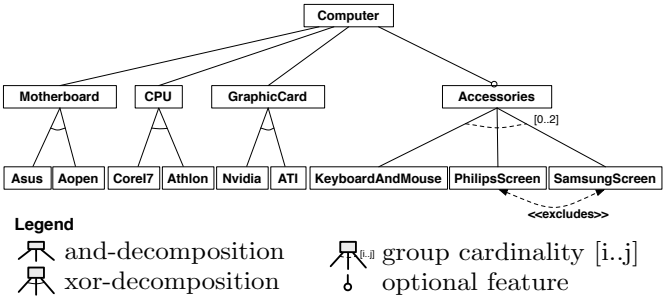

Figure 1: Computer example FD

2.1 Feature hierarchy

The TVL language has a C-like syntax: it uses braces to delimit blocks, C-style comments, semicolons to delimit statements. The rationale for this syntax choice is that nearly all computing professionals have come across a C-like syntax and are thus familiar with this style. Furthermore, many text editors have built-in facilities to handle this type of syntax.

[1] http://www.info.fundp.ac.be/~acs/tvl

In our example, the root feature, *Computer*, is decomposed into four sub-features by an *and*-decomposition: *Motherboard*, *CPU*, *GraphicCard* and *Accessories*. Furthermore, the *Accessories* feature is optional while the other three features are mandatory. In TVL, this is written as follows:

```
root Computer {
   group allOf {
      Motherboard,
      CPU,
      GraphicCard,
      opt Accessories
   }
}
```

A decomposition type in TVL is defined with the **group** keyword. Predefined decomposition operators are **allOf**, as used in this example for an *and*-decomposition, **oneOf** for *xor*-decompositions and **someOf** for *or*-decompositions. It is also possible to specify a cardinality-based decomposition with the **group [i..j]** syntax, where i and j are the lower and upper bounds of the cardinality. When defining a cardinality, one can use the asterisk character * to denote the number of children in the group, for instance **group [1..*]** would be equivalent to **group someOf**. This way, the engineer does not have to update the cardinality each time the number of children changes. Optional features like *Accessories* are declared by putting the **opt** keyword in front of their name.

FMs most commonly have a tree structure but, sometimes, a directed acyclic graph (DAG) structure – a feature can have several parents – might be useful [12]. DAG structures can also be modelled in TVL with the **shared** keyword associated to a feature name. This means that the shared feature has several parents as it is illustrated in the following example where feature D has features B and C as parents:

```
root A
   group oneOf {
      B group allOf {D},
      C group allOf {shared D}
   }
```

2.2 Attributes

In our example, the *Motherboard* has four attributes: a price, a width, an height and a socket type. TVL supports four different attribute types: integer (**int**), real (**real**), Boolean (**bool**) and enumeration (**enum**). In our example, *price*, *width* and *height* are integers. Furthermore, the *price* value is limited to values between 0 and 500. In TVL, this is expressed as follows:

```
Motherboard {
   int price in [0..500];
   int width;
   int height;
}
```

Attributes are thus declared by defining their type and name inside the definition block of the feature they belong to. Each attribute declaration is terminated by a semicolon. The **in** keyword is optional, it can be used to restrict the domain of an attribute (which might speed up automated reasoning). When declaring an attribute as an enumeration type, this means that it will take exactly one of a set of predefined values. The *socket*, for instance, is either *LGA1156* or *ASB1*. We thus declare it as an enum.

```
Motherboard {
   enum socket in {LGA1156, ASB1};
}
```

For enumerations, the **in** keyword is mandatory. Notice the use of curly braces here as opposed to square brackets for the *price* attribute above. In `TVL`, square brackets are used to declare intervals and braces to declare lists. Enumeration attributes are very similar to *xor*-decomposed features: they can be seen as a shorthand notation which avoids clutter and boilerplate code.

In many cases, the value of an attribute will be calculated based on the values of some other attributes. The value of the *price* attribute of *Accessories*, for example, is the sum of the prices of its children *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen*. Furthermore, the value of an attribute might also depend on whether its containing feature is selected or not. All this is written as follows in `TVL`:

```
Accessories {
   int price is sum(selectedChildren.price);
   group [0..2] {
      KeyboardAndMouse {
         int price is 19;
      },
      PhilipsScreen {
         int price is 99;
      },
      SamsungScreen {
         int price, ifIn: is 149, ifOut: is 0;
      }
   }
}
```

The keyword **is** can be used to set the value of an attribute, e.g. *Accessories*, *KeyboardAndMouse* and *SamsungScreen*. The keywords **ifIn:** and **ifOut:** are guards that allow to specify the value of the attribute in the case in which the containing feature is selected (**ifIn:**) or not selected (**ifOut:**). We illustrate this with the *price* attribute of the *SamsungScreen* whose value will be 149 if the feature is selected and 0 if not.

While the price of the *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen* features is fixed, the price of the *Accessories* is calculated: it is the sum (using the aggregation function **sum**) of the values of the *price* attribute of its selected children (using the **selectedChildren** keyword, which basically represents the list of values of an attribute declared in all the selected child features). Other operators are available and will be discussed in next section. A common modelling pattern for attributes declared for all features is to compute the value

of the parent feature's attribute by aggregating the attribute values of its children, up to the root. The price of a *Computer*, for example, will be calculated by summing the prices of its selected sub-features, which in turn depend on the prices of their sub-features, and so on until leaf features with fixed price values are reached.

2.3 Expressions

In TVL, expressions are used to determine the value of an attribute (as explained in the previous section) as well as to express constraints on the FM (detailed in the following section). The language is strongly typed, each expression being either of type *bool*, *real* or *int*. For more info about types see Section 4.

A basic expression is either an integer, a real, a Boolean, or a reference to a feature, an attribute or a constant. Those basic expressions can then be combined using classical operators: +, -, /, *, **abs**, for numeric values; **!**, **&&**, ||, ->, <-> for Boolean values as well as comparison operators >, >=, < or <=. Classical FM cross-tree constraints **excludes** and **requires** can also be used as Boolean expressions.

Furthermore, there are a number of aggregation functions **sum**, **mul** (multiplication), **min**, **max**, **avg** (average), **count**, **and**, **or** and **xor**. These aggregation functions can simply be used on lists of expressions or they can become powerful shorthand notations when used in combination with the **children** or the **selectedChildren** keywords. These allow to aggregate the value of an attribute that is declared for each child of a feature. The notation is **fct(children.attribute)**, or **fct(selectedChildren.attribute)** if the aggregate should be calculated on selected children only.

A full listing of the expression syntax is given in Section 3.5.

2.4 Constraints

Constraints in TVL are attached to features (classical constraints that hold 'for the whole model' can be attached to the root, for instance). They are simply Boolean expressions that can be added to the body of a feature definition, as with attribute declarations. They are terminated by a semicolon. The **ifIn:** and **ifOut:** guards we have previously seen can be used on constraints, too. In our computer example, the *Motherboard* feature has a *socket* attribute. The value of this attribute depends on the choice of the actual motherboard, i.e. on the choice of one of the sub-features. One way to model this in TVL is to define a constraint in each child feature which basically 'sets' the value of its parent's attribute.

```
Motherboard {
   enum socket in {LGA1156, ASB1};
   group oneOf {
      Asus {
         ifIn: parent.socket == LGA1156;
      },
      Aopen {
```

```
            ifIn: parent.socket == ASB1;
        }
    }
}
```

The constraint is guarded by **ifIn:**, which means that it is only applicable if the containing feature is selected.

2.5 Data blocks

`TVL` can serve as an extensible storage format for feature modelling tools. It is possible to attach to every feature a catalogue of *key/value* pairs which contain additional, tool-specific, data. If, for instance, `TVL` were to be used as the storage format for a graphical FM tool, the data block of a feature might contain the coordinates of the feature on the screen and other style information:

```
Computer {
    data {
        "xPos" "123";
        "yPos" "456";
    }
}
```

Data blocks are the only part of the language that does not have a meaning in terms of FMs. Their contents cannot be 'referenced' anywhere in the model.

2.6 Modularisation mechanisms

One of the main goals in designing `TVL` is modularity (to achieve scalability). `TVL` thus offers various mechanisms that can help users to modularise large models. First of all, custom types can be defined on at the top of the file and then be used in the FM. This allows to factor out recurring types and can thus reduce consistency errors. For instance, one might want to define the different sockets upfront and then use it as a type in an attribute declaration:

```
enum cpuSocket {LGA1156, ASB1};
...
Motherboard {
    cpuSocket socket;
}
```

It is possible to define structured types to group attributes that are logically linked. A *dimension*, for instance, is a couple (height, width) and can be declared as such using a structured type. This type can then be reused inside the *Motherboard* feature:

```
struct dimension {
    int height;
    int width;
}
...
Motherboard {
    dimension size;
}
```

Users can also specify constants using the **const** keyword followed by a type, a name and a value. These constants can then be used inside expressions or cardinalities.

```
const int maxRamBlocks 4;
```

Modularisation is achieved through two distinct mechanisms. The first is the **include** statement, which takes as parameter the path of a file (relative to the file containing the root feature). As expected, an **include** statement will include the contents of the referenced file at this point. Includes are in fact preprocessing directives and do not have any meaning beyond the fact that they are replaced by the referenced file.

```
include(./some/other/file);
```

The second mechanism is that features can be defined at one place and then extended further in the code. This has two consequences: the definition of a feature can be spread over a number of blocks and the physical hierarchy of the FM does not have to be maintained inside the code (for instance, to break up deeply nested hierarchies requiring lots of indentations).

Basically, once a feature has been defined in the group block of its parent feature, its definition can be extended any number of times. In order to extend a feature definition, one just adds a feature block with the same name to the file. This block cannot be inside another feature, it has to start its own hierarchy. Each feature block may add constraints and attributes to the feature body. The children (with the **group** keyword) can only be defined in a single one of these blocks.

This mechanism allows modellers to organise the FM according to their preferences and can be used to implement separation of concerns [13]. For example, one could separate different attribute concerns (e.g. attributes related to price and attributes related to technical details, like the sockets). Another scenario would be to declare part of the structure of the FM without detailing each feature's attributes and instead provide them later on:

```
root Computer {
   group allOf {
      Motherboard,
      CPU,
      GraphicCard,
      opt Accessories
   }
}
Computer {
   int price is sum(selectedChildren.price);
}
...
Motherboard {
   dimension size;
   ...
}
...
```

In this example, the decomposition of top feature *Computer* is defined at the beginning while its attributes and those of *Motherboard* are declared further down. The advantage of this is that the structure is easily understandable because it is not cluttered by the attributes of the different features.

3 Grammar

The grammar is a conflict-free LALR grammar. To resolve some of the conflicts, it needs operator priorities which are given in Table 2, Section 5. These priorities are not further discussed here since they are rather standard and not of interest to the discussion.

The grammar is given in extended Backus-Naur form (EBNF): terminals are enclosed in double quotes, parentheses are used for grouping, `S?` means `S` is optional, `(S)+` means that `S` repeats one or more times and `(S)*` is a shortcut for `((S)*)?`. To make the rules more readable, non-terminals are written in uppercase.

The starting non-terminal is the model, and quite naturally, a model is a sequence of type, constant and feature declarations.

```
MODEL = ( TYPE | CONSTANT | FEATURE )*
```

3.1 Type and constant declarations

A type is either a simple type (i.e. it simply renames a basic type), or a structured type with several fields. The simple types have an ID and can have a domain (declared with the **in** terminal). A structured type is just a list of simple types in curly braces, with the exception that a structured type can make use of already declared simple types (the `RECORD_FIELD = ID ID ";"` production below).

```
TYPE         =  SIMPLE_TYPE
              | RECORD ;

SIMPLE_TYPE  =  "int"  ID "in" SET_EXPRESSION ";"
              | "real" ID "in" SET_EXPRESSION ";"
              | "enum" ID "in" SET_EXPRESSION ";"
              | "int"  ID ";"
              | "real" ID ";"
              | "bool" ID ";" ;

RECORD       = "struct" ID "{" RECORD_FIELD+ "}" ;

RECORD_FIELD =  SIMPLE_TYPE
              | ID ID ";" ;
```

As expected, constants consist of the **const** terminal, their type, their identifier and their value.

```
CONSTANT =  "const" "int"  ID INTEGER ";"
          | "const" "real" ID REAL ";"
          | "const" "bool" ID ( "true" | "false" ) ";" ;
```

3.2 Identifiers

The non-terminal `ID` can refer to types, constants, but also to features and feature attributes. IDs have to start with a character and can contain numbers as well as the underscore. There are two rules that are not formalised in the grammar and will be detailed in Section 4: feature IDs have to start with an uppercase letter and IDs in `TVL` are case sensitive.

```
ID = ("a"-"z" | "A"-"Z")
     ("a"-"z" | "A"-"Z" | "0-9" | "_") * ;
```

In order to allow feature names to occur several times, and to allow references to an attribute of a feature inside the body of another feature, it is possible to build chains of IDs separated with a dot. All IDs of a chain with at least two IDs, except for the last one, have to denote features. The last element may be a feature or an attribute. In this context, the terminals **root**, **this** and **parent** help to make specifications more intuitive, as seen in Section 2.4.

```
SHORT_ID =  "root"
          | "this"
          | "parent"
          | ID ;

LONG_ID  =  SHORT_ID
          | SHORT_ID "." LONG_ID ;
```

3.3 Feature declarations

A feature declaration consists of an ID (optionally preceded by the **root** terminal) which is followed either by its body (that is, a number of `FEATURE_BODY_ITEM`s), or directly by its children block (the `FEATURE_GROUP` non-terminal) if one wants to omit attribute or constant declarations. In case a feature is extended rather than declared for the first time, it may use a long ID since the feature being extended might not have a unique name. When extending the root this way, the **root** terminal must be omitted.

```
FEATURE =  "root" ID "{" FEATURE_BODY_ITEM+ "}"
         | "root" ID FEATURE_GROUP
         | LONG_ID "{" FEATURE_BODY_ITEM+ "}"
         | LONG_ID FEATURE_GROUP ;
```

The feature body consists of several items which can be data blocks, constraints, attributes or the group block declaring the child features.

```
FEATURE_BODY_ITEM =  DATA
                   | CONSTRAINT
                   | ATTRIBUTE
                   | FEATURE_GROUP ;
```

The children body starts with the group terminal, followed by the cardinality and the list of children. A cardinality can be either a predefined one, or an interval of natural numbers which can be specified directly, with natural numbers, constants (the `ID` non-terminal) or the asterisk character as explained in Section 2.1.

```
FEATURE_GROUP = "group" CARDINALITY "{"
                    HIERARCHICAL_FEATURE
                    ("," HIERARCHICAL_FEATURE)*
                "}" ;

HIERARCHICAL_FEATURE =  ( "opt" )? FEATURE
                      | ( "shared" | "opt" )? LONG_ID ;

CARDINALITY =  "oneof"
             | "someof"
             | "allof"
             | "[" (ID | NATURAL | "*" ) ".."
                   (ID | NATURAL | "*" ) "]" ;
```

3.4 Attributes and constraints

An attribute declared for a feature is either a single attribute (`BASE_ATTRIBUTE`) or a structured attribute. A structured attribute has to be of a structured type defined previously (the first `ID`), have a name (the second `ID`) and may list the fields of that structured type to define a body for them. A single attribute is just a type followed by a name and, optionally, the body.

```
ATTRIBUTE      =  BASE_ATTRIBUTE
                | ID ID "{" SUB_ATTRIBUTE+ "}" ;

BASE_ATTRIBUTE =  "int"  ID ATTRIBUTE_BODY? ";"
                | "real" ID ATTRIBUTE_BODY? ";"
                | "bool" ID ATTRIBUTE_BODY? ";"
                | "enum" ID ATTRIBUTE_BODY? ";"
                | ID     ID  ATTRIBUTE_BODY? ";" ;

SUB_ATTRIBUTE  = ID ATTRIBUTE_BODY ";" ;
```

The attribute body allows to restrict the domain of an attribute, or to give it a value as part of the attribute declaration (instead of doing it in the constraints). A fixed value is defined with the **is** terminal. A domain is specified with the **in** terminal followed by a `SET_EXPRESSION` (second and third production). In this case, one can further specify a conditional value assignment (which does not make sense in the case the attribute value is fixed). Such a conditional value assignment can also be specified alone, i.e. without being preceded by an **is** or **in** statement.

```
ATTRIBUTE_BODY =  "is" EXPRESSION
                | "in" SET_EXPRESSION
                       ( "," ATTRIBUTE_CONDITIONNAL ) ?
                | "," ATTRIBUTE_CONDITIONNAL ;
```

A conditional value assignment allows to specify an **is** or an **in** statement that should hold depending on whether the feature in which the attribute is declared is selected (**ifin:** terminal) or not (**ifout:** terminal). If both cases are given, they have to be separated by a comma and should start with the **ifin:** terminal. The productions simply capture the eight possible combinations for this.

```
ATTRIBUTE_CONDITIONNAL =
   "ifin:"  "is" EXPRESSION
                  "," "ifout:" "is" EXPRESSION
 | "ifin:"  "is" EXPRESSION
 | "ifout:" "is" EXPRESSION
 | "ifin:"  "in" SET_EXPRESSION
                  "," "ifout:" "is" EXPRESSION
 | "ifin:"  "is" EXPRESSION
                  "," "ifout:" "in" SET_EXPRESSION
 | "ifin:"  "in" SET_EXPRESSION
                  "," "ifout:" "in" SET_EXPRESSION
 | "ifin:"  "in" SET_EXPRESSION
 | "ifout:" "in" SET_EXPRESSION ;
```

A constraint declaration is just a boolean expression terminated by a semicolon. Just as an attribute value declaration, it can be guarded with the **ifin:** or **ifout:** terminals.

```
CONSTRAINT =  EXPRESSION ";"
            | "ifin:" EXPRESSION ";"
            | "ifout:" EXPRESSION ";" ;
```

3.5 Expressions

The expression syntax is meant to be as complete as possible in terms of operators, to encourage writing of intuitive constraints. The meaning of most productions should be clear.

```
EXPRESSION =
   (* Reference to a feature/an attribute *)
   LONG_ID

   (* Grouping *)
 | "(" EXPRESSION ")"

   (* Classical FM constraints *)
 | LONG_ID "excludes" LONG_ID
 | LONG_ID "requires" LONG_ID

   (* Conditional expression *)
 | EXPRESSION "?" EXPRESSION ":" EXPRESSION

   (* Boolean *)
 | EXPRESSION "&&"  EXPRESSION
 | EXPRESSION "||"  EXPRESSION
 | EXPRESSION "->"  EXPRESSION  (* implication *)
 | EXPRESSION "<-"  EXPRESSION
 | EXPRESSION "<->" EXPRESSION  (* equivalence *)
 | "!" EXPRESSION
 | "true"
 | "false"

   (* Boolean aggregation *)
 | "and" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
 | "or"  "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
 | "xor" "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
```

```
  (* Comparison *)
| EXPRESSION "==" EXPRESSION
| EXPRESSION "!=" EXPRESSION
| EXPRESSION "<=" EXPRESSION
| EXPRESSION "<"  EXPRESSION
| EXPRESSION ">=" EXPRESSION
| EXPRESSION ">"  EXPRESSION

  (* Domain restriction *)
| EXPRESSION "in" SET_EXPRESSION

  (* Arithmetic *)
| EXPRESSION "+" EXPRESSION
| EXPRESSION "-" EXPRESSION
| EXPRESSION "/" EXPRESSION
| EXPRESSION "*" EXPRESSION
| "-" EXPRESSION
| "abs" "(" EXPRESSION ")"
| INTEGER
| REAL

  (* Arithmetic aggregation *)
| "sum"   "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "mul"   "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "min"   "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "max"   "(" (EXPRESSION_LIST | CHILDREN_ID) ")"
| "count" "(" ("children" | "selectedchildren") ")"
| "avg"   "(" (EXPRESSION_LIST | CHILDREN_ID) ")" ;
```

The **in** statements that can be used to define or restrict the domain of an attribute require a `SET_EXPRESSION` that specifies the set of values. A set expression can either be a list of expressions inside curly braces (i.e. the set is defined *in extension*) or an interval with integer, real or infinite (the ***** character) bounds between square brackets (to define a set *in intension*).

```
SET_EXPRESSION =
   "{" EXPRESSION_LIST "}"
 | "[" (INTEGER | REAL | "*" ) ".."
       (INTEGER | REAL | "*" ) "]" ;
```

An expression list is just a comma-separated list of expressions. Its main use is in defining sets in extension, but it can also be used in combination with an aggregation function. The domain of an enum (i.e. the values of an enum) is actually an expression list where every expression is an `ID` non-terminal.

```
EXPRESSION_LIST = EXPRESSION ("," EXPRESSION_LIST)* ;
```

To concisely specify cases in which the value of an attribute is an aggregate of another attribute that is declared for each child, the **children** statement can be used (followed by a `LONG_ID` denoting the attribute).

```
CHILDREN_ID =  "selectedchildren" "." LONG_ID
             | "children" "." LONG_ID ;
```

3.6 Data blocks

A data block starts appropriately with the **data** terminal, followed by a key/-value list in curly braces where key and value are separated by a blank and each pair is terminated by a semicolon. There can be several data blocks in each feature, all being merged when the diagram is parsed. Keys should be unique for each feature. As noted before, data keys do not have any meaning in the normal FD semantics.

```
DATA  = "data" "{" DATA_PAIR+ "}" ;
DATA_PAIR = STRING STRING ";" ;
```

3.7 Values

For completeness sake, we also give the rules for naturals, integers, reals and strings which are rather standard. Strings have to be enclosed in double quotes; double quotes can be escaped with a backslash.

```
NATURAL = "0" | ["1"-"9"]["0"-"9"]* ;
INTEGER = "0" | ("-")?["1"-"9"]["0"-"9"]* ;
REAL    = INTEGER "." (["0"-"9"]*["1"-"9"])? ;
STRING  = '"' [^] '"' ;
```

4 Well-formedness

There are a number of rules a `TVL` model has to adhere to in order to be valid.

4.1 Naming, scope and references

The naming rules for names of features, attributes, types, constants, enum values and struct fields are similar to other C-like languages: they can use letters, digits, the underscore and cannot begin with a digit; names are case-sensitive. Furthermore, there is a list of reserved keywords that may not be used.

There is one mandatory naming convention, namely that feature names must begin with a capital letter while names of attributes, types, constants and struct fields must begin with a lowercase letter (enum values are exempt from this rule). This prevents a number of potential naming conflicts. Of course, names have to be unique within their scope, that is,

- child features and attributes of the same feature,
- all declared types,
- constants,
- struct fields that are siblings,
- enum values per enum type.

are all required to have distinct names. Furthermore,

- no enum value may have the name of an attribute, a feature or a constant
- no constant may have the name of an attribute.

Feature and attribute names, however, do not have to be globally unique except for the name of the root feature. To reference a feature called *bar* inside a constraint, one can just use its name, i.e. write *bar*, or use a *qualified name*. Assuming its parent feature to be called *foo*, a qualified name would be *foo.bar*, or *baz.foo.bar* if *baz* is the parent of *foo*; and so on. A *fully qualified name* is one that starts with the name of the root feature.

Each reference to a feature inside a constraint has to be unambiguous. This means that if the name of the referenced feature is unique, it is sufficient to put this name. Otherwise, an unambiguous qualified name has to be used, that is, the feature name has to be prefixed by the names of its parents until the uppermost name is unique. Since the name of the root feature has to be unique, a fully qualified name is always unambiguous.

The rules for the referencing of attributes are similar. Inside the body of their containing feature they can be referenced solely with their name. Otherwise they have to be prefixed by the name of their containing feature. If this name is ambiguous, the same rules as above apply.

There are a number of keywords to make referencing easier: **parent** denotes the parent feature of the feature in the body of which it is used, **root** always denotes the root feature and **this** denotes the feature in the body of which it is used.

4.2 Type correctness

`TVL` is strongly typed, and does not allow type casting, furthermore, `TVL` type checking can be performed statically. Type correctness is defined as expected: expressions defining the value of an attribute have to be of the same type as the attribute. Constraints have to be expressions of type *bool*. The expressions themselves have to be correctly typed, that is, Boolean operators may only take Boolean operands, numeric operators may only take numeric operands, and so on.

When a set is defined in extension, i.e. with a list of elements, all elements in the list need to have the same type, which is the type of the set. Expressions involving attributes of type *enum* may only use enum values defined for the attribute.

4.3 Other rules

The decomposition relation of the model has to be acyclic. While the grammar allows to declare a cycle in the decomposition relation, this is not permitted. Decomposition cardinalities $\langle i, j \rangle$ have to be so that $i \leq j$ and i has to be smaller than or equal to the number of child features. There can only be one **group**

block per feature. The **parent** keyword may not be used for features having more than one parent.

The **children** keyword can be used in combination with an aggregation function to apply the function to the value of the attribute of all children of the feature. Its use therefore requires that the attribute is indeed declared for all the children of the feature, that the children all declare it with the same type and that this type is compatible with the aggregation function. The **selectedChildren** keyword is similar, except for the fact that it ranges only over children that were selected. The same rules apply here. In addition, this keyword cannot be used for the **min** and **max** aggregation functions and it can only be used if the decomposition relation of the parent feature is so that there will always be at least one selected child.

In the following section, we consider that models are compliant with all those well-formedness rules.

5 Semantics

In line with previous work of the authors [6, 14], a language is not fully defined without a formal semantics. Fortunately, part of the work has already been done elsewhere, mainly by Schobbens *et al.* [6] with the formal definition of Free Feature Diagrams (FFD), a parameterised FM language.

However, we cannot reuse the FFD definition as is. FFD are based on an abstract syntax that is much more limited than the concrete syntax of `TVL`. In Section 5.1, we thus define a translation from `TVL` to an abstract syntax close to that of FFD. Furthermore, FFD do not formalise attributes or non-Boolean constraints. Also, they do not explicitly capture the notion of *optional* feature, which they encode with an intermediate dummy feature that is $\langle 0..1 \rangle$-decomposed. We contribute these missing pieces in Section 5.2.

For the definition of the semantics, we follow the guidelines of Harel and Rumpe [15], meaning that we formally define the abstract syntax $\mathcal{L}$ of our language, the semantic domain $\mathcal{S}$ and the semantic function $\mathcal{M} : \mathcal{L} \to \mathcal{S}$.

5.1 Abstract syntax $\mathcal{L}_{\texttt{TVL}}$

The concrete syntax introduced in the previous section offers a number of syntactic shortcuts (structuring mechanisms, types,...). In order to obtain an easily formalisable language, the abstract syntax for `TVL` will be that of a *normal form* with less constructs but equal expressiveness.

Definition 1 (`TVL` Abstract Syntax, extension of [6])**.** *The syntactic domain $\mathcal{L}_{TVL}$ is the set of all tuples $(N, r, DE, \omega, \lambda, A, \rho, \tau, V, \iota, \Phi)$ where:*

- $\mathbf{N}$ *is the (non empty) set of features,*
- $\mathbf{r} \in N$ *is the root,*

- **DE** $\subseteq N \times N$ *is the decomposition (hierarchy) relation between features. For* $(n, n') \in DE$, n *is the parent and* n' *the child feature. For convenience, we will sometimes write* $n \rightarrow n'$ *instead of* $(n, n') \in DE$,
- $\omega : N \rightarrow \{0, 1\}$ *labels optional features with a* 1,
- $\lambda : N \rightarrow \mathbb{N} \times \mathbb{N}$ *indicates the decomposition operator of a feature, represented as a cardinality* $\langle i..j \rangle$ *where* i *is the minimum number of children required in a configuration and* j *the maximum (we use angle brackets to distinguish cardinalities from other tuples),*
- **A** *is the set of attributes,*
- $\rho : A \rightarrow N$ *is a total function that gives the feature declaring the attribute,*
- $\tau : A \rightarrow \{int, real, enum, bool\}$ *assigns a type to each attribute,*
- V *is the set of possible values for enumerated attributes,*
- $\iota : \{a \in A | \tau(a) = enum\} \rightarrow \mathcal{P}(V)$ *defines the domain of each enum,*
- $\mathbf{\Phi} \subseteq \mathcal{L}_{exp}$ *is a set of Boolean-valued expressions over the features* N *and the attributes* A*, expressing additional constraints on the model.* $\mathcal{L}_{exp}$ *is the set of all correctly typed Boolean-valued expressions* B *that are formed according to the grammar given in Table 1, where* $n \in N$ *is a feature,* $a \in A$ *is an attribute,* $d \in \mathbb{Z}$ *is an integer,* $q \in \mathbb{Q}$ *is a rational number,* t *is an enum value and* $v \in \mathbb{V}$ *is an enum value.*

Furthermore, each $d \in \mathcal{L}_{\texttt{TVL}}$ *must satisfy the following well-formedness rules:*

- r *is the unique root* $\forall n \in N(\nexists n' \in N \bullet n' \rightarrow n) \Leftrightarrow n = r$,
- r *is not optional* $\omega(r) = 0$,
- *DE is acyclic* $\nexists n_1, .., n_k \in N \bullet n_1 \rightarrow .. \rightarrow n_k \rightarrow n_1$,
- *Terminal nodes are* $\langle 0..0 \rangle$*-decomposed.*

We recall that the abstract syntax, $\mathcal{L}_{\texttt{TVL}}$, only covers a subset of the concrete `TVL` syntax defined in Section 3. A `TVL` model using only constructs from $\mathcal{L}_{\texttt{TVL}}$ is in *normal form*, and the subset of the `TVL` language reduced to models in normal form is called $\texttt{TVL}_{NF}$. In the following, we will show that any `TVL` model

Table 1: Expression syntax $\mathcal{L}_{exp}$ of $\mathcal{L}_{\texttt{TVL}}$

B ::= `true` | `false` | n | a | v | E `in` S |
n `excludes` n | n `requires` n |
B `&&` B | B `||` B | `!`B |
B `->` B | B `<-` B | B `<->` B |
E `==` E | E `!=` E |
E `<=` E | E `<` E | E `>=` E | E `>` E |
`and(`B [`,` B]* `)` | `or(`B [`,` B]* `)` |
`xor(`B [`,` B]* `)`

E ::= n | a | t | d | q |
E `+` E | E `-` E | E `/` E | E `*` E | `-` E |
`abs(`E`)` | B `?` E `:` E |
`sum(`E [`,` E]* `)` | `mul(`E [`,` E]* `)` |
`min(`E [`,` E]* `)` | `max(`E [`,` E]* `)`

S ::= `{` E [`,` E]* `}` |
`[` (d | `*`) `..` (d | `*`) `]` |
`[` (f | `*`) `..` (f | `*`) `]`

Table 2: Operator precedence in TVL and $\texttt{TVL}_{NF}$

Associativity	Operators
right	**!**, (unary) **-**, and aggregation functions
non	**requires**, **excludes**
left	*, /
left	+, -
non	>, <, >=, <=
non	==, ! =, **in**
left	&&
left	||
non	<->
left	->
right	<-
right	**? :**

can be transformed into an equivalent $\texttt{TVL}_{NF}$ model. The semantics of the TVL language is thus provided in two steps. A first step is to provide a formal semantics to the many constructs and syntactic shortcuts that are not part of $\texttt{TVL}_{NF}$ by giving a syntactic translation from TVL to $\texttt{TVL}_{NF}$. The second step is to define the semantics of $\texttt{TVL}_{NF}$, i.e. that of $\mathcal{L}_{\texttt{TVL}}$.

The concrete syntax of $\texttt{TVL}_{NF}$ is a subset of the concrete syntax of TVL. The only allowed constructs are those defining the features and their hierarchy (N, r and DE), optional features (ω), cardinality-based decomposition operators (λ) and attributes with basic types (A, ρ and τ). The excluded constructs are mainly the structuring mechanisms and the non-cardinality decomposition operators. Furthermore, constraints in $\texttt{TVL}_{NF}$ (Φ) have to be expressions of $\mathcal{L}_{exp}$. To obtain an expression in $\mathcal{L}_{exp}$ from a $\texttt{TVL}_{NF}$ expression, we have to define operator precedence, associativity and parentheses, since Table 1 abstracts away from these. We chose to define operator precedence in TVL and $\texttt{TVL}_{NF}$ to be the same as in C.

Definition 2 (Operator precedence in TVL and $\texttt{TVL}_{NF}$)**.** *Table 2 lists all operators in decreasing order of precedence. The associativity of each operator is given in the left column. Parentheses can be used to group expressions and force a different evaluation order.*

Now, the remaining constructs map 1:1 to the elements of $\mathcal{L}_{\texttt{TVL}}$ and $\mathcal{L}_{exp}$ in Definition 1. The first part of the semantics is provided in Definition 3 which specifies how to translate constructs that only exist in TVL into $\texttt{TVL}_{NF}$, thereby defining their semantics.

Definition 3. *A model in $\texttt{TVL}_{NF}$ is obtained from a model in TVL by applying the following transformation steps in the specified order.*

Includes. Eliminate all `include` preprocessing directives by replacing them with the content of the referenced files.

Constants. Eliminate constants `const t c e;` by replacing all occurrences of `c` by its definition `e`.

Types. Here we distinguish between types that merely *rename* basic types `b t;` and more complex *structured* types `struct t {b1 t1, b2 t2,..}`. The former can be eliminated by replacing all occurrences of the defined type `t` by the corresponding basic type `b`. Structured types can be eliminated in a two-step process. The first step is to flatten a structured type `t` by replacing it by a number of individual types `b1 t_t1; b2 t_t2;..`, and to flatten attributes declared as structs by replacing them by individual attributes. The flattened types are then eliminated in a recursive step.

Attribute domain and value specifications. The construct `t a in s;` allows to specify the range of an attribute `a` to be the set `s`. The `in` construct is removed and a constraint of the form `this.a in s;` is added. Similarly, the construct `t a is v;` allows to specify a fixed-value attribute `a` to be `v`. The `is` construct is removed and a constraint of the form `this.a == v;` is added.

Conditional domain and value specifications. An attribute value or domain specification can also be guarded with the keywords `ifIn:` and `ifOut:`, the syntax then is `t a, ifIn: vin, ifOut: vout;` where `vin` can be `in s` to specify a domain or `is v` to specify a value. These constructs are removed and constraints of the form `ifIn: this.a == v; ifOut: this.a == v; ifIn: this.a in s;` or `ifIn: this.a in s;` are added.

Guards. Guarded constraints `ifIn: c;` and `ifOut: c;` are replaced by equivalent constraints `this -> (c);` and `!this -> (c);` respectively.

Aggregation with comprehension. Eliminate the keywords `children` and `selectedChildren` as follows, assuming that $c_1, .., c_k$ are the child features of the containing feature:

- Replace `avg(children.a)` by `sum(children.a) / count(children)`, and similarly for `selectedChildren`.
- Replace `fct(children.a)` by `fct(`c_1`.a, ..., `c_k`.a)`, where `fct` is one of the aggregation functions `sum`, `mul`, `min`, `max`, `and`, `or`, `xor`.
- Replace `count(children)` by the number of children of the feature.
- Replace `count(selectedChildren)` by `sum((`c_1` ? 1 : 0), ..., (`c_k` ? 1 : 0))`.
- Replace `fct(selectedChildren.a)` by `fct((`c_1` ? `c_1`.a : neut), ..., (`c_k` ? `c_k`.a : neut))`, where `fct` is one of the aggregation functions `sum`, `mul`, `and`, `or`, `xor`, and `neut` is the neutral element wrt. the aggregation function (i.e. 0 for addition, 1 for multiplication, *true* for conjunction and $false$ for disjunction and xor). Remember that the `selectedChildren` keyword for these functions is only available if the parent decomposition enforces the selection of at least one feature.

Relative names. All relative names `parent`, `this` and `root` are resolved and replaced by unambiguous feature names.

Constraints. A single set of constraints is obtained in `TVL` by moving all constraints to the root feature.

Decomposition operators. First replace occurrences of `oneOf`, `allOf` and `someOf` by `group [1..1]`, `group [*..*]` and `group [1..*]` respectively. In a second step, replace each occurrence of `*` inside a cardinality by the number of child features.

Distributed definitions. Gather feature definitions spread over different blocks into the single block inside the group statement of its parent.

This translation will effectively eliminate all constructs that are not in $\texttt{TVL}_{NF}$.

5.2 Semantics of $\texttt{TVL}_{NF}$

The semantic domain defines the universe in which an element of the syntactic domain is to be interpreted [15]. As in the existing definition by Schobbens *et al.* [6], the semantic domain is that of *product lines*, meaning that a given FM should be interpreted as a product line. In earlier definitions, a product line is formally defined as a *set of products*, and a product as a *set of features*. While this definition is still relevant in our case, it does not capture the notion of *attribute*. We thus redefine a product as a set of features that comes with a function providing a value for each attribute.

Definition 4 (Semantic domain $\mathcal{S}$)**.** *The* semantic domain *of `TVL`, denoted $\mathcal{S}$, is the set of all products, each product p being a couple $p = (c, v)$ where c is a set of features and v is a valuation of the attributes, respecting τ and ι, formally:*

$$\mathcal{S} = \mathcal{P}(\mathcal{P}(N) \times \mathcal{P}(A \rightarrow \mathbb{Z} \cup \mathbb{Q} \cup \{true, false\} \cup V))$$

One could argue that the attributes should in fact not be part of the semantic domain, that they are just helpers to express additional constraints between features. While this is a valid interpretation for FMs, it will preclude a number of possibilities offered by attributes, essentially further reasoning or filtering based on the values of the attributes (such as attribute optimisation [7, 16]).

Basically, each attribute is treated like a variable that is always defined, even if the feature that declares it is not part of the product. We chose this interpretation as its flexible and emphasises the constraint-language aspect of FMs. An alternative interpretation would have been to assume that attributes of non-selected features do not exist (as if they were not declared). This would lead to several problems: what to do with attributes defined in terms of attributes that do not exist because their parents are not in the product, or: what is the semantics of constraints over undeclared attributes. Moreover, it would also cause problems when considering the semantics of FM configuration [17], where in an intermediate state some features are selected, some deselected and some undecided.

Given the semantic domain from Definition 4, the semantic function describes how to interpret each element of the syntactic domain from Definition 1.

Definition 5 (Semantic function $\mathcal{M}$)**.** *Given a* TVL *model* $d \in \mathcal{L}_{TVL}$*, its semantics is given by the function* $\mathcal{M} : \mathcal{L}_{TVL} \to \mathcal{S}$*, where* $\mathcal{M}(d)$ *is the set of all couples* (c, v) *with* $c \in \mathcal{P}(N)$ *being a* valid feature set *and* $v : A \to \mathbb{Z} \cup \mathbb{Q} \cup \{true, false\} \cup V$ *being a* valid attribute valuation*. Each* $(c, v) \in \mathcal{M}(d)$ *is such that:*

- *c contains the root:* $r \in c$*;*
- *c satisfies decomposition cardinality:*

$$\begin{array}{l} \forall f \in c \bullet \lambda(f) = \langle m..n \rangle \\ \quad \Rightarrow \ m - |opt_N| \leq |mand_c| \\ \quad \wedge \ |all_c| \leq n \\ \textit{where: } opt_N = \{g | g \in N \ \wedge \omega(g) = 1 \ \wedge \ f \to g\} \\ \qquad\quad mand_c = \{g | g \in c \ \wedge \omega(g) = 0 \ \wedge \ f \to g\} \\ \qquad\quad all_c = \{g | g \in c \ \wedge \ f \to g\} \end{array}$$

- *c includes each selected feature's parent:*

$$\forall g \in c \bullet f \to g \Rightarrow f \in c$$

- *c and v satisfy all the* $\phi \in \Phi$*, meaning that* $\forall \phi \in \Phi \bullet [\![\phi]\!](c, v) \not\models false$*. The semantics of an expression,* $[\![\phi]\!](c, v)$*, is quite standard and given in Table 3.*

While one might think that optional features are rather easy to formalise, the existing formal semantics by Schobbens *et al.* [6] only covers them indirectly (with syntactic preprocessing). Moreover, existing semantic discussions such as those by Czarnecki and Eisenecker [18] are limited to the interplay between optional features and standard *and*-, *or*- and *xor*-decompositions. As noted in [18], if one child of an $\langle 1..j \rangle$-decomposed feature f is optional, then this is equivalent to all its children being optional, or to all its children being mandatory and f being $\langle 0..j \rangle$-decomposed. A similar observation holds for a $\langle 1..1 \rangle$-decomposed feature with at least one optional child. This appears to cause confusion to the point that existing tools generally support optional features only as children in an *and*-decomposition.

Intuitively, optionality has 'priority over' the decomposition relation: an *and*-decomposition mandates that all features be included if their parent is, yet optional features are not bound by this requirement. Our definition generalises this intuition to the case of arbitrary $\langle i..j \rangle$ cardinalities. As can be seen in the second point of Definition 5, optional features cause the lower bound of a decomposition cardinality to decrease by the number of optional features opt_N. This alone would be incorrect; in addition, the features counted to satisfy the lower bound are only the mandatory features of the configuration $mand_c$. The latter part is best illustrated with an example, consider:

```
root f group [3..3] {
   a, opt b, c
}
```

Table 3: Expression semantics in $\texttt{TVL}_{NF}$, that is, the value of $[\![\phi]\!](c, v)$.

$[\![\texttt{true}]\!] = true$	$[\![\texttt{and(}\mathrm{B}_1\texttt{, .., }\mathrm{B}_k\texttt{)}]\!] = true$ iff $\bigwedge_{i\in[1,k]}[\![B_i]\!]$
$[\![\texttt{false}]\!] = false$	$[\![\texttt{or(}\mathrm{B}_1\texttt{, .., }\mathrm{B}_k\texttt{)}]\!] = true$ iff $\bigvee_{i\in[1,k]}[\![B_i]\!]$
$[\![\mathrm{n}]\!] = true$ iff $n \in c$	$[\![\texttt{xor(}\mathrm{B}_1\texttt{, .., }\mathrm{B}_k\texttt{)}]\!] = true$ iff $\bigoplus_{i\in[1,k]}[\![B_i]\!]$
$[\![\mathrm{a}]\!] = v(a)$	$[\![\mathrm{E}_1\ \texttt{+}\ \mathrm{E}_2]\!] =$ the value of $[\![\mathrm{E}_1]\!] + [\![\mathrm{E}_2]\!]$
$[\![\mathrm{a}\ \texttt{==}\ \mathrm{t}]\!] = true$ iff $v(a)$ equals t	$[\![\mathrm{E}_1\ \texttt{-}\ \mathrm{E}_2]\!] =$ the value of $[\![\mathrm{E}_1]\!] - [\![\mathrm{E}_2]\!]$
$[\![\mathrm{a}\ \texttt{!=}\ \mathrm{t}]\!] = true$ iff $v(a)$ does not equal t	$[\![\mathrm{E}_1\ \texttt{/}\ \mathrm{E}_2]\!] =$ the value of $[\![\mathrm{E}_1]\!] \,/\, [\![\mathrm{E}_2]\!]$
$[\![\mathrm{E}\ \texttt{in}\ \mathrm{S}]\!] = true$ iff $[\![\mathrm{E}]\!] \in [\![\mathrm{S}]\!]$	$[\![\mathrm{E}_1\ \texttt{*}\ \mathrm{E}_2]\!] =$ the value of $[\![\mathrm{E}_1]\!] * [\![\mathrm{E}_2]\!]$
$[\![\mathrm{n}_1\ \texttt{requires}\ \mathrm{n}_2]\!] = true$ iff $n_1 \notin c \vee n_2 \in c$	$[\![\texttt{-E}]\!] =$ the value of $-[\![\mathrm{E}]\!]$
$[\![\mathrm{n}_1\ \texttt{excludes}\ \mathrm{n}_2]\!] = true$ iff $n_1 \notin c \vee n_2 \notin c$	$[\![\texttt{abs(E)}]\!] =$ the absolute value of $[\![\mathrm{E}]\!]$
$[\![\mathrm{B}_1\ \texttt{\&\&}\ \mathrm{B}_2]\!] = true$ iff $[\![\mathrm{B}_1]\!] \wedge [\![\mathrm{B}_2]\!]$	$[\![\mathrm{B}_1\ \texttt{?}\ \mathrm{E}_1\ \texttt{:}\ \mathrm{E}_2]\!] =$ if $[\![\mathrm{B}_1]\!]$, then $[\![\mathrm{E}_1]\!]$, otherwise $[\![\mathrm{E}_2]\!]$
$[\![\mathrm{B}_1\ \texttt{||}\ \mathrm{B}_2]\!] = true$ iff $[\![\mathrm{B}_1]\!] \vee [\![\mathrm{B}_2]\!]$	$[\![\texttt{sum(}\mathrm{E}_1\texttt{, .., }\mathrm{E}_k\texttt{)}]\!] =$ the value of $\sum_{i\in[1,k]}[\![E_i]\!]$
$[\![\texttt{!B}]\!] = true$ iff $[\![\mathrm{B}_1]\!]$ equals $false$	$[\![\texttt{mul(}\mathrm{E}_1\texttt{, .., }\mathrm{E}_k\texttt{)}]\!] =$ the value of $\prod_{i\in[1,k]}[\![E_i]\!]$
$[\![\mathrm{B}_1\ \texttt{->}\ \mathrm{B}_2]\!] = true$ iff $[\![\mathrm{B}_1]\!] \Rightarrow [\![\mathrm{B}_2]\!]$	$[\![\texttt{min(}\mathrm{E}_1\texttt{, .., }\mathrm{E}_k\texttt{)}]\!] =$ the smallest value of the $[\![E_i]\!]$
$[\![\mathrm{B}_1\ \texttt{<-}\ \mathrm{B}_2]\!] = true$ iff $[\![\mathrm{B}_2]\!] \Rightarrow [\![\mathrm{B}_1]\!]$	$[\![\texttt{min(}\mathrm{E}_1\texttt{, .., }\mathrm{E}_k\texttt{)}]\!] =$ the greatest value of the $[\![E_i]\!]$
$[\![\mathrm{B}_1\ \texttt{<->}\ \mathrm{B}_2]\!] = true$ iff $[\![\mathrm{B}_1]\!] \Leftrightarrow [\![\mathrm{B}_2]\!]$	$[\![\texttt{\{ }\mathrm{E}_1\texttt{, .., }\mathrm{E}_k\texttt{ \}}]\!] =$ the set $\{[\![E_i]\!] \mid i \in [1,k]\}$
$[\![\mathrm{E}_1\ \texttt{==}\ \mathrm{E}_2]\!] = true$ iff $[\![\mathrm{E}_1]\!]$ equals $[\![\mathrm{E}_2]\!]$	$[\![\texttt{[}\mathrm{d}_1\ \texttt{..}\ \mathrm{d}_2\texttt{]}]\!] =$ the interval $[d_1, d_2]$
$[\![\mathrm{E}_1\ \texttt{!=}\ \mathrm{E}_2]\!] = true$ iff $[\![\mathrm{E}_1]\!]$ does not equal $[\![\mathrm{E}_2]\!]$	$[\![\texttt{[* .. d]}]\!] =$ the interval $]-\infty, d]$
$[\![\mathrm{E}_1\ \texttt{<}\ \mathrm{E}_2]\!] = true$ iff $[\![\mathrm{E}_1]\!] < [\![\mathrm{E}_2]\!]$	$[\![\texttt{[d .. *]}]\!] =$ the interval $[d, +\infty[$
$[\![\mathrm{E}_1\ \texttt{>}\ \mathrm{E}_2]\!] = true$ iff $[\![\mathrm{E}_1]\!] > [\![\mathrm{E}_2]\!]$	$[\![\texttt{[}\mathrm{f}_1\ \texttt{..}\ \mathrm{f}_2\texttt{]}]\!] =$ the interval $[f_1, f_2]$
$[\![\mathrm{E}_1\ \texttt{<=}\ \mathrm{E}_2]\!] = true$ iff $[\![\mathrm{E}_1]\!] \leq [\![\mathrm{E}_2]\!]$	$[\![\texttt{[* .. f]}]\!] =$ the interval $]-\infty, f]$
$[\![\mathrm{E}_1\ \texttt{>=}\ \mathrm{E}_2]\!] = true$ iff $[\![\mathrm{E}_1]\!] \geq [\![\mathrm{E}_2]\!]$	$[\![\texttt{[f .. *]}]\!] =$ the interval $[f, +\infty[$

In that case, valid products are $\{f, a, b, c\}$ and $\{f, a, c\}$. If only the lower bound were decreased, $\langle 2..3\rangle$, then the products $\{f, a, b\}$ and $\{f, b, c\}$ would be considered valid as well. This is why in Definition 5 the number of selected *mandatory* children of f has to be greater than the new lower bound.

The concept of feature attribute is also not formally defined in the existing literature. As discussed above, feature attributes exist independently of the feature that declares them. Our definition is purely declarative, it just requires that the attribute values satisfy all constraints. Such a definition lends itself well to implementation in SAT or CSP solvers. Furthermore, we chose not to fix attributes to a default value in case their parent feature is not part of the product. Basically, the same constraints apply to attributes whether their parent feature is selected or not (since the model is just one big constraint). Otherwise, it would be impossible to give fixed values to attributes (such as, the price of a feature). Furthermore, TVL provides appropriate syntactic sugar:

```
root f group allOf {
   opt a {
      int i, ifIn: in [1..10], ifOut: is 0;
      int j is 42;
      int k;
   }
```

```
}
```

Here, the value of the attribute `i` is between one and ten if `a` is in the product, and zero otherwise. The attribute `j` is fixed at 42 and `k` can take any value.

Acknowledgements

This work was partially funded by the Walloon Region, the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy under the MoVES project, the BNB and the FNRS.

References

[1] A. Classen, Q. Boucher, and P. Heymans, "A text-based approach to feature modelling: Syntax and semantics of TVL," *To appear in Science of Computer Programming, Elsevier*, 2010.

[2] Q. Boucher, A. Classen, P. Faber, and P. Heymans, "Introducing TVL, a text-based feature modelling language," in *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*. University of Duisburg-Essen, January 2010.

[3] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[4] A. Classen, P. Heymans, and P.-Y. Schobbens, "What's in a feature: A requirements engineering perspective," in *Proceedings of FASE'08*, 2008, pp. 16–30.

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, CMU, Tech. Rep., 1990.

[6] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Feature Diagrams: A Survey and A Formal Semantics," in *Proc. of RE'06*, 2006.

[7] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés, "Automated reasoning on feature models," in *Proceedings of CAiSE'05*, 2005.

[8] D. Benavides, D. Batory, and P. Grünbacher, Eds., *Proceedings of VaMoS'10*, ser. ICB Research Report, vol. 37. Universität Duisburg-Essen, 2010.

[9] L. Chen, M. A. Babar, and N. Ali, "Variability management in software product lines: A systematic review," in *Proceedings of SPLC'09*, 2009, pp. 81–90.

[10] A. Hubaux, A. Classen, M. Mendonca, and P. Heymans, "A preliminary review on the application of feature diagrams in practice," in *Proceedings of VaMoS'10*, January 2010, pp. 53–59.

[11] A. Hubaux, Q. Boucher, H. Hartman, R. Michel, and P. Heymans, "Evaluating a text-based feature modelling language: Four industrial case studies," in *Accepted for publication at the 3rd International Conference on Software Language Engineering (SLE 2010)*, July 2010.

[12] K. C. Kang, S. Kim, J. Lee, K. Kim, G. J. Kim, and E. Shin, "Form: A feature-oriented reuse method with domain-specific reference architectures," *Ann. Softw. Eng.*, vol. 5, pp. 143–168, 1998.

[13] P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *Proceedings of ICSE'99*, 1999.

[14] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen, "Evaluating formal properties of feature diagram languages," *IET Software Journal, Special Issue on Language Engineering*, vol. 2, no. 3, pp. 281–302, 2008.

[15] D. Harel and B. Rumpe, "Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff," Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, Tech. Rep., 2000.

[16] T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans, "Relating requirements and feature configurations: A systematic approach," in *Proceedings of SPLC'09*, 2009.

[17] A. Classen, A. Hubaux, and P. Heymans, "A formal semantics for multi-level staged configuration," in *Proceedings of VaMoS'09*, 2009, pp. 51–60.

[18] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.